Fast Linking of Separately-Compiled FPGA Blocks without a NoC

Yuanlong Xiao, Syed Tousif Ahmed, and André DeHon
Dept. of Electrical and Systems Engineering, University of Pennsylvania, Philadelphia, PA, USA
Email: ylxiao@seas.upenn.edu, stahmed@seas.upenn.edu, andre@ieee.org

2020 International Conference on Field-Programmable Technology (ICFPT) | 978-1-6654-2302-1/20/$31.00 ©2020 IEEE | DOI: 10.1109/ICFPT51103.2020.00035

***Abstract*—Dedicated point-to-point wires (DW) can be used in place of a Packet-Switched Networks-on-a-Chip (PSNoC) for fast linking of separately-compiled FPGA blocks, providing higher bandwidth and performance with less area overhead without increasing compile time. Previous work showed that separate compilation of FPGA modules using a pre-compiled FPGA overlay could reduce the long FPGA compile time by defining and separately mapping small partially reconfigurable blocks (Processing Elements) and using a fixed PSNoC to connect them together. Nonetheless, the lightweight PSNoC cannot meet the high data transmission requirements for some critical links, limiting overall performance. We demonstrate that DWs, where the producer ports and consumer ports are directly connected instead of sharing limited-throughput, packet-switched connections, can provide us with high throughput between Processing Elements (PEs) while preserving the fast compile time; the DWs also reside on partially reconfigurable blocks and can be compiled along with the reconfigurable PEs simultaneously on the cloud. Adjacent pages can be connected by fast links with low latency. Mapping Rosetta Benchmarks, we show that the application-customized direct networks can offer 1.5–10× performance gain and 47–86% interface area overhead savings compared to previous work with PSNoCs.**

***Index Terms*—FPGA, packet-switched, direct wire, nearest-neighbor, overlay, compile time, divide-and-conquer**

I. Introduction

The huge resource flexibility and energy efficiency of FPGAs provide new opportunities for not only compute-bounded computation, but also a large class of memory-bounded applications [1]. Instead of spending months to years on architecture design and tape-out verification, FPGAs supply the designers with instant implementation by only downloading bitstreams into an FPGAs within minutes. With mature High-Level Synthesis (HLS) front-end technology, FPGA-based applications can be developed in high-level program language like C/C++, OpenCL or Python, leading to higher coding productivity. However, the back-end tools lag behind these versatile front-end tools. Compiling the design into bitstreams often takes hours. HLS and logic synthesis can be run in separate threads independently on different blocks, but placement and routing exploits limited parallelism with commodity EDA tools. This means we cannot make full use of the abundant cloud computing resources to accelerate the placement and routing directly. This long compile time limits the efficiency for initial debugging and incremental refinements, which eventually prevents more developers from embracing FPGAs.

Park [2] proposes to decompose large designs into small, separate units, and uses Packet-Switched Network-on-a-Chip (PSNoC) to connect them together. These units can be compiled in parallel, and only a specific unit needs to be re-compiled when changes are made to it. Xiao [3] evaluates this method with concrete instances, and develops a tool called *PRFlow*. By mapping the Rosetta Benchmarks [4] to the XCZU9EG (ZCU102) [5], Xiao shows the compilation time can be reduced from hours to 12 minutes. However, the performance is often much lower than the monolithic SDSoC implementation, and the worst-case benchmark is 12.5 times slower than the SDSoC version. While the pre-compiled PSNoC can perform quick initial implementation, the limited bandwidth between separately-compiled blocks can greatly harm the performance. We propose: *it is not necessary to sacrifice a significant amount of the raw FPGA bandwidth (performance) to achieve shorter compile times.*

In this paper, we investigate using direct, pipelined wires to replace PSNoCs. Instead of sharing only one physical port into PSNoCs, different blocks can be directly connected by dedicated physical links. With dedicated interconnect, the user throughput can be improved from 9.6 Gbps per module [3] to 97.2 Gbps. By using Relay Stations [6], [7], the stream links can be pipelined to guarantee timing closure. By making the network blocks reconfigurable, the dedicated interconnection can be compiled in parallel with the user logic, which adds no extra compile time compared to Xiao [3] and Park's [2] work. The simple network reduces interconnect latency and overhead.

We make the following key contributions:

- Characterize the routing capability in regions of the FPGA and relate it to packet-switched NoC bandwidth (Sec. III-D)
- Demonstrate the potential of direct-wire switchbox routing to reduce compilation time compared to monolithic design mapping without sacrificing performance (Sec. V)
- Show potential to unify logic and switching partial reconfiguration regions (Sec. IV)

II. Background

A. Accelerating FPGA Compile Time

Previous work that investigates reducing compile time can be found in [2], [3], [8]–[12]. Lavin [8] uses RapidSmith [13] to reduce compile time by saving implementation (synthesis,

978-1-6654-2302-1/20/$31.00 ©2020 IEEE
DOI 10.1109/ICFPT51103.2020.00035

placement, and route) data in the form of hard macros, and connecting these macros together by a design stitcher. However, HMFlow only links up pre-compiled hard macros and requires a final linking phase that still takes time. Moreover, the essential intermediate Xilinx-generated files, like XDL (Xilinx Design Language) files and NCD (Netlist Circuit Description) files, are no longer supported by Vivado, which makes HMFlow unusable for devices after 7-Series. RapidWright [14] updates RapidSmith to support Vivado designs, but only provides a low-level interface to customize Vivado checkpoints and build special-purpose placement and routing tools. It is a potentially useful backend to build upon, but does not, itself, support general, automated compilation acceleration. Just In Time (JIT) compilation [10] provides the users with Domain Specific Language (DSL) to develop FPGAs, which can link the design patterns with pre-compiled bitstreams at runtime. The overlay, on top of the FPGAs, connects all the reconfigurable tiles similar to the switchboxes, but it only routes word-wide data. Like HMFlow, JIT can only speed up certain design patterns (Innner Product, Matrix Multiply, Correlation) where the overlay blocks have been pre-designed and does not consider incremental hardware refinements. *Seiba* [11] provided an FPGA overlay consisting of HLS generated circuits, an execution manager, and soft-processor function unit. The execution manager can move some functionality from HLS-generated RTL circuits to the soft-processor incrementally without hardware recompilation. Seiba is a complementary technique and does not support parallel acceleration of the native FPGA compiles; it cannot evaluate the hardware accelerators performance without regenerating the whole overlay, which is essential for incremental FPGA development within a short edit-compile-debug cycle.

B. NoC Overlayed on FPGAs

With the advent of FPGAs with high-density and regular reconfigurable gates, researchers have explored NoC design on top of commercial FPGAs. As interconnect wires can be shared by different PEs, the NoC can reduce the number of dedicated wires required and save area. By pipelining and clocking the linking wires independently of the PEs, PEs can be isolated within certain areas, increasing the frequency and throughput of the communication links. When the throughput between different PEs is low compared to the PE's operating cycle, different PEs can share the limited bandwidth, and a PSNoC can allocate the bandwidth dynamically to where it is needed. Compared to dedicating wires to links that are idle, it might provide higher bandwidth to the active links when they need it. A variety of PSNoCs are designed on FPGAs. The CMU CONNECT [15] can automatically generate the RTL code for NoCs for synthesis. It can achieve comparable or better performance than the publicly available RTL-level NoC code [16], with only one-half FPGA resources, or provide 3–4$\times$ performance gain under the same area cost. The Penn split-merge design showed how to pipeline the switches to achieve higher bandwidth [17]. However, they did not consider the floorplan, which can in turn affect the PE implementations. Packet-Switched and Time-Multiplexed networks are explored in [18], giving us some guidance on how to make best use of different networks. These Packet-Switched NoCs spend considerable resources on packet buffers. Recently proposed bufferless, deflection-routed NoCs are more compact without sacrificing performance [19], [20] and variants support continued operation during partial reconfiguration [21].

C. Logic Emulators

Logic emulators [22], [23] previously solved the problem of decomposing large designs into separate components and linking them together with overlay partial crossbars [22] or statically time-multiplexed networks [23]. However these logic emulators did not achieve the fast compiles that we achieve and sacrificed one to two orders of magnitude of raw FPGA performance, while this work show how to maintain most of the FPGA performance while keeping 10–18 minute compile times.

D. Partial Reconfiguration

Partial Reconfiguration (PR) is an FPGA technology that allows only parts of the FPGA design to be reconfigured while keeping other parts untouched [24]. With this feature, the pre-defined parts can even be reconfigured during runtime, while the other parts can run as normal. As the partial bitstream is smaller than the complete one, it is often used to reduce the reconfiguration time. The standard procedures for Xilinx PR are to pre-define the reconfigurable parts as pblocks and define the unchanged parts as static region. PR is widely used in reducing area [25], [26], [27], decreasing bitstream loading time, and reducing the compilation time [2], [3]. PR has also been used to configure a 928$\times$928 crossbar in [28]. Our approach operates at a lower level than [28], constructing only dedicated paths directly on the FPGA fabric instead of paying the higher cost of generating a complete crossbar.

E. Latency-Insensitive Dataflow Model

We target designs developed for the streaming computing model [7], [29]–[31]. The whole design is decomposed into individual operators that are connected together by latency-insensitive stream links [32]. The streaming link adds *ready* and *valid* control signals to the raw data. When the *valid* and *ready* are both asserted, the data are transmitted from the producer to the consumer. As the directions of *valid* and *ready* are different, we use Relay Station [6], [33] to pipeline the stream links.

F. Fast Mapping with PRFlow

We build upon the work of Park and Xiao [2], [3]. *PRFlow* is a tool developed to accelerate the FPGA compile time with Partial Reconfiguration (PR) technique. They propose to divide one FPGA chip into separate, small partial reconfigurable blocks, called leaves or pages. These pages are connected by a fixed, packet-switched network. The deflection-routed, packet-switched, Butterfly Fat Tree (BFT) [34] network is adopted,

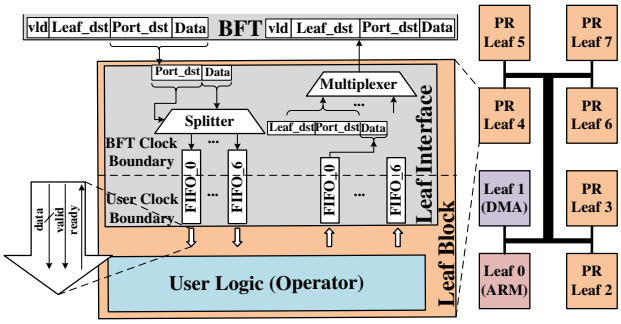

Fig. 1: BFT NoC and Leaf Interface

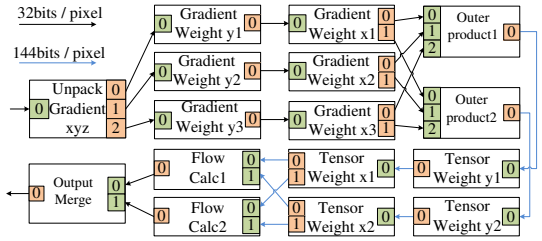

Fig. 2: Optical Flow Dataflow Graph

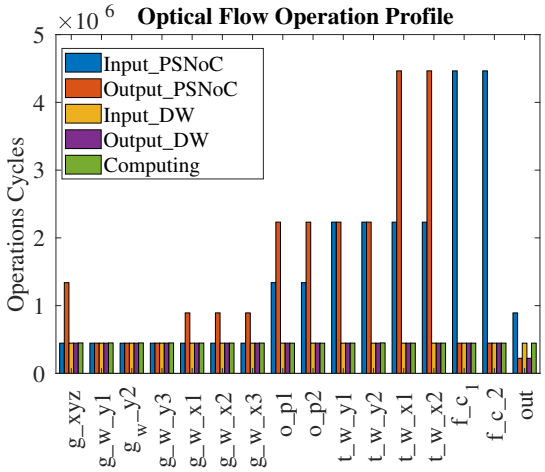

Fig. 3: Optical Flow IO Profile

as it is a lightweight PSNoC for modern FPGA architectures [20]. The applications can be developed in the form of small latency-insensitive operators, connected by stream links, as described in Sec. II-E. The operators can be mapped and compiled to PR blocks in parallel on the cloud, with a mapping time around 12 minutes, while never needing to compile the complete design together. As long as the packet-switch network is placed and routed, we only need to configure the source and destination registers inside each page to link the pages together.

III. Idea

In this section, we first identify two problems with the PSNoC overlay: *Bandwidth Waste in PRFlow* and *Interface Sharing Logic* area overhead. Next, we characterize the routing capability in FPGAs. With that background, we introduce and elaborate our direct wire idea and its supporting techniques (relay stations, partition pins). Finally, we discuss exploiting nearest-neighbor links to further utilize the on-chip bandwidth and reduce latency.

A. Problem: Bandwidth Waste in PRFlow

Similar to most of the PSNoCs, in Xiao's overlay, each PE has a uniform interface [3]. On the BFT side, only one pair of physical 32-bits IO buses, are implemented, which can save resources, but degrade the performance due to 9.6 Gbps (300MHz $\times$ 32bits) throughput between pages and BFT. This partly explains how some benchmark implementations lose performance compared to the monolithic mapping (See Tab. V). For the *Rosetta optical flow* benchmark (Figs. 2 and 3), some pages require multiple cycles to send and receive data, but only require one cycle to process data.

B. Profile IO Throughput

We can profile the read/write operations for all the pages for each benchmark. As the overall performance is determined by the maximum computing cycles or maximum IO operation cycles, we normalize the computing cycles and IO cycles as below.

$$NormComputeCycles = \frac{ComputeCycles}{Max\{AllComputeCycles\}} \quad (1)$$

$$NormIoCycles = \frac{Max\{InputCycles, OutputCycles\}}{Max\{AllComputeCycles\}} \quad (2)$$

We plot the ratio between $NormComputeCycles$ and $NormIoCycles$ for all the operators in our benchmark set in Fig. 4. The black line means that the IO operation cycles equal the maximum computing cycles. If the ratio points are below the black line, the IO operations cycles will not affect the overall performance, as the application is still compute-bound. When the points are above the black line, and their X-axis values are far smaller than 1.0, it means the computing cycles are small, but the high IO operating cycles significantly limit the performance, like the *face detection* and *spam filter* benchmarks. We should also address those benchmarks that have high ratio values, like the *optical flow*. However, we can see most of the IO-compute ratios are around or below 1.0 in Fig. 5, which means only a limited number of links in each application need high bandwidth interconnections.

C. Problem: Interface Sharing Logic

Similar to the TCP/IP protocol, data is transmitted through a PSNoC in packets. Extra acknowledgment logic must be added to producers and consumers to support a windowed acknowledgment discipline [35] to guarantee that the input FIFO has enough space to accept data from the PSNoC. Since

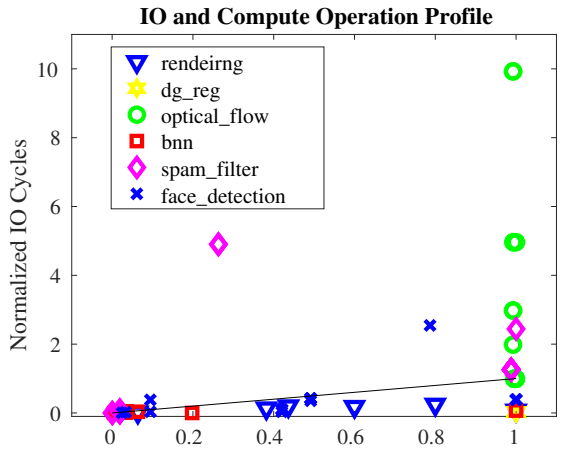

Fig. 4: IO Profile for all Benchmarks

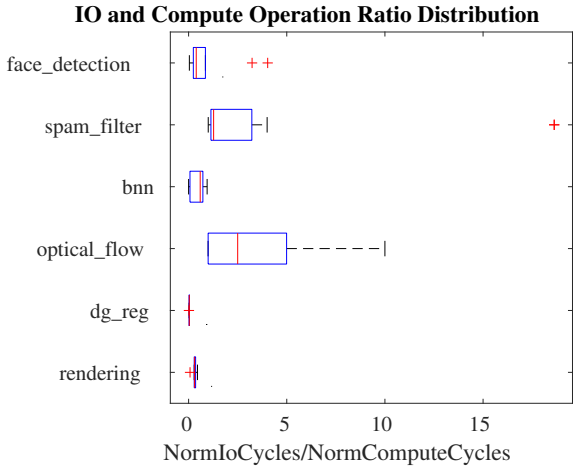

Fig. 5: IO Compute Ratio for all Benchmarks

data transmissions in deflection-routed PSNoCs are out-of-order, special order label headers are added into each packets, which also increase the wire overhead. Xiao [3] uses a 48b flit to send 32b of data, consuming one-third of the PSNoC leaf bandwidth on header overhead. The resource consumption of PSNoC leaf interface is shown in Tab. I. The output ports have dedicated FIFOs and multiplexers, meaning designs will consume resources proportional to the number of output ports. The input ports only need to connect the input data to the corresponding port according to the headers. The total logic LUT count overhead of leaf interface is 577–2821, occupying 10%–56% of the 5,000–6,000 LUT Pages used by Xiao [3] according to the number of ports (Tab. I).

Since the PSNoC designs are often limited by page throughput as noted above, Xiao [3] runs the PSNoC at a higher frequency than the leaves to maximize bandwidth. This requires an asynchronous FIFO in the leaf interface to cross between the network and leaf clock boundaries. Unfortunately, the asynchronous FIFO adds 7 cycles of latency to both the

TABLE I: PSNoC Leaf Interface Resource Consumption

IO Number	Sub-module	LUT	FF	BRAM18
1	in ports	121	196	5
	out ports	172	224	1
	control_logic	284	318	0
3	in ports	363	588	15
	out ports	557	672	3
	control_logic	442	424	0
5	in ports	595	980	25
	out ports	944	1120	5
	control_logic	538	518	0
7	in ports	833	1372	35
	out ports	1436	1568	7
	control_logic	552	613	0

connection to and from the network, meaning every link in the design has at least 14 cycles of added latency compared to the monolithic design. While the added latency is not an issue for feed-forward pipelines, it can have a significant impact on performance in the case where there are cyclic dependencies in the flow graph.

D. Opportunity: FPGA Wire Density

Our basic idea is to use direct-wire interconnect to replace packet-switched networks. From Section III-A and III-B, we know the bandwidth between the PSNoC and the pages is the bottleneck. While we could increase the PSNoC data width, it will also increase the overall PSNoC area overhead linearly. To decrease the data transmission cycles in Fig. 3, we need to increase the PSNoC datawidth by 9× to meet the throughput requirements for the 8 blue links in Fig. 2. We propose to customize the datawidth for only those 8 blue links. The dedicated wires directly connect the producer ports and consumer ports together; since the ports do not need to be serialized onto the PSNoC, it maximizes the throughput. The raw metal wires in modern, island-style FPGAs are abundant, suggesting we can support many connections without networks to share wires [36]. For example, there are around 104 wires in the horizontal directions and 200 wires in the vertical directions for each Configurable Logic Blocks (CLBs) in Zynq UltraScale+ series chips [37]. As the height of each page is 60 CLBs, there are 60×200=12,000 wires available to route data out of the page. Since the direct-wire topology is simple, we expect the EDA tools can easily route the interconnections by using more of those raw metal wires inside FPGAs. *As the network and the user logic can all be defined as reconfigurable blocks, they can be compiled in parallel.*

We do expect we will need to use the raw wires sparsely for the designs to be routable. To explore how many raw wires we can use in FPGAs, we conduct a routing capacity exploration. We incrementally increase the stream ports between network blocks and page blocks to explore how many wires can be used between these two (vertically across 60 CLBs). We also use the same method to explore the routing capability between network blocks. In Tab. II, we report a coarse-grained, design-space exploration on datawidth, W, and frequency, F. Our optimization object is the product of W and F, which can offer us the highest throughput. From Tab. II, we can see the

TABLE II: Static Timing Analysis Slack for 60 CLB Boundary

W \ F(MHz)	100	200	250	300	400
32	3.761	0.861	X	X	X
64	1.187	0.284	0.287	X	-0.853
96	3.151	0.438	-0.764	X	X
128	X	X	X	0.021	-1.009
160	3.292	0.536	X	X	-0.905
192	3.262	X	X	X	X
224	X	X	X	-0.01	X
256	X	0.431	0.071	0.002	X
288	3.262	X	X	0.009	X

Numbers in cells represent the slack; slack less than zero fail to meet the timing target. X means the routing cannot be completed.

maximum on-chip throughput is 86.4 Gbps (288×300MHz)—9× higher than the peak bandwidth available in the PSNoC.

E. Direct Wiring

In Fig. 1, we could see all the user IOs share one BFT IO bus pair. For the direct wires, we connect the ports directly with dedicated pipeline stream wires. For example, to map the dataflow graph in Fig. 6a to our DW overlay in Fig. 6b, all 4 of the links out of operator *a* get their own, dedicated wires as they would in the monolithic compilation case instead of sharing one physical port in PSNoC. The pages and switchboxes are all defined as reconfigurable pblocks. We can define the connections in the direct-wire switchboxes automatically based on the connectivity of the design. We use our own Python scripts to generate switchbox Verilog files according to the connection needs of a particular application. To guarantee timing closure, we add relay stations (Sec. II-E) in each switch modules. The direct wires can also ensure in-order data transmission without extra flow control logic.

F. Partition Pins

While we can support additional input and output wires to pages as identified in Tab. II, they do have a cost. Vivado demands the allocation of partition pins for each IO from the PR regions. With default Vivado placement, these partition pins can reduce routability and render BRAMs and DSPs in the region unusable.

This has some consequence on the selection of page regions. It is best to avoid placing DSP and BRAM columns directly on the edge of the page facing the switchbox. Columns of CLBs at the edge will provide more partition pin locations that are less likely to interfere with BRAM and DSP use.

In the case of DSPs/BRAMs, when partition pins are placed in CLB blocks immediately adjacent to DSP/BRAM blocks, they can block the cascade connections between DSPs/BRAMs making the DSPs/BRAMs unusable for typical configurations with cascade connection. We can reduce this effect by synthesizing DSP blocks without cascades (synthesis option: `-cascade_dsp tree`), but this likely comes at the cost of increasing latency for the computation. BRAM cascades can be tuned by specifying the cascade height at the RTL level (property: CASCADE_HEIGHT), but cannot be completely disabled.

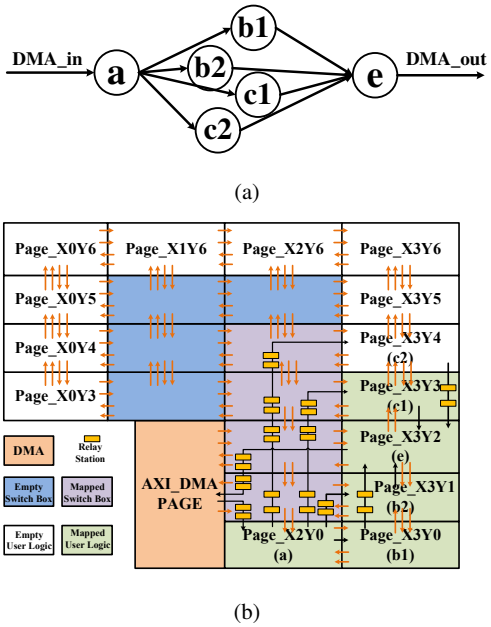

Fig. 6: (a) Application Dataflow Graph. (b) Mapping Application to Direct-Wire Architecture

In the worst-case, effective page capacity must be recharacterized as a function of the allocated partition pins.

G. Relay Station and Interface

The PSNoC of Xiao adds flow control logic to prevent packet loss when operators cannot keep up with data and uses addresses in flits to tolerate packet reordering (Sec. III-C). This further forces the interface to be instantiated in pairs. Our dedicated interconnect constructs real paths between different ports. It needs more resources when mores links need to be implemented, but it can eliminate the flow control interface. The network overhead for PSNoC is fixed for each page at around 500 LUTs. For the DW, the wires are determined by different applications. Assuming we use all the pages, and all the link need 3 relay stations, the equivalent overhead can afford 8 32-bits direct links ((2×500 LUTs per interface)/(40 LUTs/relay-station×3 relay-stations)) between two pages. By using a simple interface with an asynchronous FIFO to replace the PSNoC flow control leaf interface, the interface overhead per link can be reduced from 576 LUTs and 738 FFs to 310 LUTs and 422 FFs. Since we do not need to run the network at a different clock frequency from the leaf pages, we can use a simple interface with synchronous FIFOs to replace the PSNoC flow control leaf interface, reducing the interface overhead to only 184 LUTs and 154 FFs per link and also decreasing the latency from 14 cycles (7×2) to 4 cycles (2×2). We can use a single FIFO on one end, cutting the resource overhead in half and reducing the added latency to 2 cycles.

From Fig. 7, LUT consumption for both the packet switching and direct wires increase linearly with the IO number.

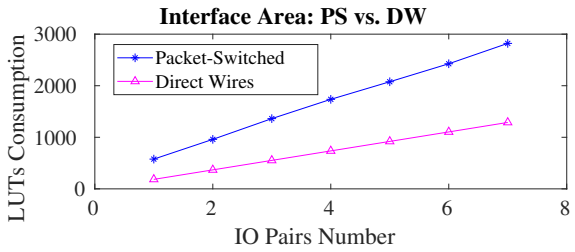

Fig. 7: Interface LUT Consumption Comparisons

TABLE III: Direct Wires Overhead

	Datawidth	LUT	FF	BRAM18
Relay Station	32	40	73	0
Async Interface	32	155	211	1
Sync Interface	32	92	77	1

We can see the direct wires can save 54–68% LUT overhead compared to the packet-switched network depending on the number of IOs, and it can offer exclusive throughput for each port.

H. Nearest-Neighbor Interconnect

Replacing the PSNoC with DW, we can fully utilize the raw wires between switchboxes and pages, but this only leverages the bandwidth on one edge—the edge adjacent to the switchbox. It does not exploit the other 3 edges of each page. The DW links need to go up to one switchbox before arriving at the destination pages, which still adds additional latency even when the two connected pages are physically adjacent to each other. To address the bandwidth and latency issue above, we can also add direct interconnect between adjacent pages, which can help not only further improve the throughput, but also reduce the latency between pages. In Fig. 6b, we can see page_X3Y1 has one boundary adjacent to switchbox, and 2 boundaries with page_X3Y2 and page_X3Y0. We can add short, neighbor interconnect as the yellow arrows show. For example, in Fig. 6, more links (a→b1, b2→e, c1→e) can be mapped by nearest-neighbor interconnects. This can further increase the throughput on top of the maximum throughput we explore in Section III-D. Additionally, these quick interconnect links can transmit data with short latency.

IV. Universal Pages

Once we accept that we are potentially remapping both the leaf compute pages and the switchbox partial reconfigurable pblocks (Sec. III-D), it becomes enticing to consider if we can unify the two kinds of separately-compiled, partial reconfiguration regions into a single, fungible resource. That is, can we divide the chip up into a number of "pages" and then populate those pages with logic, interconnect, or even some combination between them?

A. Idea

Starting from our nearest-neighbor tree design, we could imagine decomposing each of our switchboxes into smaller switchbox pages (Fig. 8). This would create even smaller routing tasks for each of the smaller switchboxes, potentially accelerating their mapping time. In the simplest case, we use this just as before with more decomposition; we map each switchbox page with the necessary direct-wire linkage required to connect the design. However, we could also choose to reallocate one of the switchbox pages to a compute page. For example, in a design that admits to more nearest-neighbor connections, we could "borrow" switchbox page SWB 2d as a compute page that primarily communicated by nearest-neighbor connections to compute page 23 and 19. Alternately, in a design that required more routing, we could reallocate page 25 to serve as an extra switchbox page. Generalizing, universal pages allow us to tune the compute and switchbox page ratio. We were motivated, in part, to explore this universal page design due to challenges of developing a single overlay that worked well across a set of benchmarks. The ability to customize interconnect allocation eased the overlay design.

Page 31	SWB 3d	Page 27	Page 23	SWB 2d	Page 19
Page 30	SWB 3c	Page 26	Page 22	SWB 2c	Page 18
Page 29	SWB 3b	Page 25	Page 21	SWB 2b	Page 17
Page 28	SWB 3a	Page 24	Page 20	SWB 2a	Page 16
SWB Ta	SWB Tb	SWB Tc	SWB Td	SWB Te	SWB Tf
Page 3	SWB 0a	Page 7	Page 11	SWB 1a	Page 15
Page 2	SWB 0b	Page 6	Page 10	SWB 1b	Page 14
DMA	SWB 0c	Page 5	Page 9	SWB 1c	Page 13
ARM	SWB 0d	Page 4	Page 8	SWB 1d	Page 12

Fig. 8: Decomposed Switchboxes toward Universal Pages (compare to Fig. 9)

We could go one step further and "share" a uniform page between logic and switching. In the simplest case, we might put a small compute operator in with a switchbox that is not heavily congested. Switchbox page SWB 2d might easily hold some small combining logic, such as a reduce operation that we see in several designs.

B. Prior Work

The idea of merging interconnect and logic is not new and dates back, at least, to channelless or Sea-of-Gates gate arrays. The NYU UltraComputer pushed the idea of integrating logic into the communication network [38]. In the FPGA field, Triptych [39], and to some extent, cellular arrays like CAL [40], which later became the Xilinx 6200 [41], embraced the strategy. More recently, this has been explored in the Amorphous FPGA [42] and Liquid Silicon [43].

C. Costs

This does require the addition of more partition pins at the sub-switchbox or universal page boundaries, which impact

routing and logic usability as seen in Sec. III-F. If fully embraced, it means any pages shared between interconnect and switching may need to be recompiled as interconnect changes. This still admits parallelism, but may increase the number of pages that must be recompiled when the design changes.

V. Demonstration

A. Methodology

To evaluate the impact of our ideas, we realize our DW interconnect and nearest-neighbor links on top of PRFlow [3]. We use *Vivado and SDSoC 2018.2* as the EDA tool and map Rosetta benchmarks [4] to the UltraScale+ Zynq XCZU9EG (274,080 LUTs, 548160 Flip-Flops, 1824 18Kb BRAMs, and quad ARM Cortex-A53 CPU). We perform the compilation on the Google Cloud. Each compute node is equipped with 4 dual-thread, 2.8 GHz Intel Xeon Cascade Lake processors and 64GB RAM; Vivado runs with 8 threads.

B. Benchmark Refinement

Following Xiao [3], we also use Rosetta Benchmark source (https://github.com/cornell-zhang/rosetta, commit ID 6bc38c0), but we further refine the benchmarks and apply the revisions to both the PSNoC and DW implementations.

1) 3D Rendering: We reduce the total pages from 12 to 6 by merging some small pages.

2) Digit Recognition: We keep the hamming distance calculation operators unchanged, but split the final sorter into 20 decomposed operators. Instead of transmitting the final 120 minimum distance candidates into a central, final sorter, we arrange for the 20 decomposed operators to perform a systolic minimum reduce, with each operator taking the minimum of its own candidates and the minimum candidates passed from its neighbor.

3) SPAM Filter: No changes are made for this benchmark.

4) Optical Flow: As we use DW, the input and output data do not need to queue up and share the physical BFT ports, meaning the initiation interval (II) can be restored to one (same as SDSoC version).

5) Binarized Neural Network (BNN): Following Xiao, we also store all the BNN parameters on the on-chip-BRAM, but we resize the BRAM size according to our different page size and decrease the number of pages from 29 to 18.

6) Face Detection: To fit the 5K page-size, Xiao [3] decomposed the integral images and line buffers into 5 parts, and replace the unroll pragma with pipeline pragma. This is reasonable as the PSNoC cannot supply enough bandwidth between pages, but we restore the unroll pragma, since the DWs and nearest-neighbor wires can provide enough inter-page bandwidth.

C. PSNoC

The PSNoC overlay is shown in Fig. 9. We divide the packet-switched network into 5 blocks, and distribute the switches into different blocks. All the switchbox blocks are defined as reconfigurable.

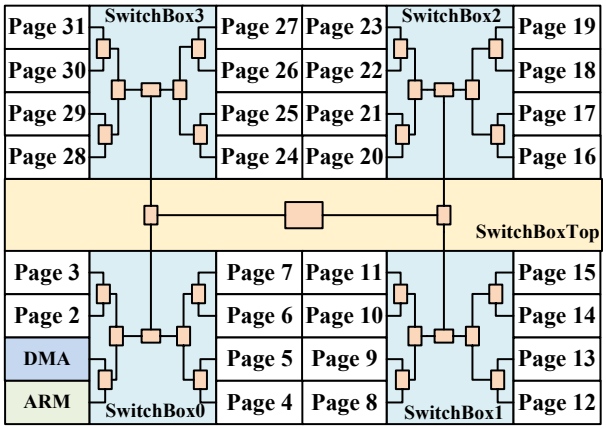

Fig. 9: PSNoC Overlay Floorplan

D. Universal Overlay Design

We organize our overlay with universal pages as shown in Fig. 11. The white boxes are nominal user logic pages, and the blue boxes are the nominal switchbox pages. As shown in Fig. 11b, only adjacent pages have interconnection wires, unifying the direct wires (black) and the nearest-neighbor wires (yellow). We have 14 pure user pages, but when more pages are needed, we can also put some operators into switch box pages. For example, for *face detection*, we also map the user logic into the blue switch pages along with the connection wires. The biggest challenge is to choose the proper number of wires to allocate between different user/switch pages. We run all the benchmarks, get the minimum used wires numbers for each benchmark, and use a superset of all the wiring requirements, so that this one-size-fits-all overlay can map all the benchmarks. To make the overlay more generic, we set the minimum number of wires to 130 in order to get at least 97.24 Gbps throughput for each page (130 $\times$ 187MHz $\times$ 4 edges). The detailed information for the overlay is shown in Fig. 11a. The page sizes are identical within a vertical column.

E. Application Mapping to Universal Overlay

We assign the stream operators to different pages according to the page size. We need to put operators with high data transmission requirements in adjacent pages to take advantage of the higher bandwidth and lower latency of nearest-neighbor links. In Fig. 10, we can map *rendering* and *spam filter* easily, but we need to borrow logic from switch boxes to map complex examples like *face detection*, *BNN*, *digit recognition*, and *optical flow*.

F. Performance

By mapping the Rosetta Benchmarks [4], we tabulate the main performance benefits in Table V. The second column is the SDSoC runtime. By using Vivado+SDK, we can customize the DMA engine, and the third column lists the performance with our customized DMA engine. The PS and DW use the same customized DMA engines. We rerun the refined

TABLE IV: Application Resources

Benchmark	Resource	Mono. (no BFT)	PS User	PS Leaf Interface	PS Route	PS Total	Direct Wire (our work) User	Direct Wire (our work) LI	Direct Wire (our work) Route	Direct Wire (our work) Total
Rendering	LUTs	12750	6162	5084	50400	61646	6365	1854	78905	87124
	BRAMs	97	78	60	288	426	78	16	458	552
	DSPs	0	0	0	516	516	0	0	816	816
Digit Reg	LUTs	32556	37229	6266	50400	93895	33586	3262	70648	107496
	BRAMs	384	320	120	288	728	320	80	384	784
	DSPs	0	0	0	516	516	0	0	816	816
Spam Filter	LUTs	12931	6498	20220	50400	77118	6974	5514	77366	89854
	BRAMs	136	108	252	288	648	108	54	465	628
	DSPs	224	256	0	516	772	256	0	816	1072
Optical Flow	LUTs	79146	22114	12287	50400	84801	22056	3851	71052	96959
	BRAMs	186	84	138	288	510	84	58	461	693
	DSPs	252	280	0	516	798	282	0	682	964
BNN	LUTs	46165	10143	19878	50400	80421	8812	2664	74331	85807
	BRAMs	1198	933	228	288	1449	920	24	312	1256
	DSPs	3	6	0	516	522	3	0	814	817
Face Detection	LUTs	55849	97477	20876	50400	168753	110618	6170	39662	158450
	BRAMs	211	159	276	288	723	192	80	379	661
	DSPs	78	102	0	516	618	144	0	686	830

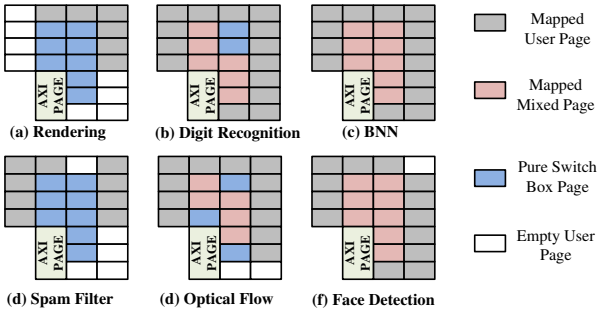

Fig. 10: Rosetta Benchmark Page Assignment

TABLE V: Application Performance

Benchmark	SDSoC	Mono. (no BFT)	PS	DW (our work) with NN
Rendering	1.5	1.4	1.2	1.3
Digit Reg	6.9	5.0	10.8	5.4
Spam Filter	28.2	22.4	48.9	32.2
Optical Flow	3.5	2.1	25.8	2.6
BNN	5.3	3.6	17.4	22.4
Face Detection	18.2	24.3	101.0	33.1

Results are throughput in time (ms) per frame or input.

benchmarks (Sec. V-B) from Xiao [3], and list the results in column four; these are typically higher performance than [3]. From the fifth column, we can see the DW interconnect can greatly improve over the PS case performance by 1.2–12×. For the *optical flow*, we only increase the bandwidth for the critical links (discussed in Section III-D), but we can increase the performance by 12×. For the *face detection*, we can improve the performance by 3×.

As shown in Fig. 4, the *BNN* is compute limited rather than IO limited; consequently, it is not improved simply by the move to DW. The monolithic design exploits some optimizing pragmas that the current decomposition is not able to exploit due to page resource restrictions. The additional IO bandwidth from DW will permit different decompositions that may admit further optimization that can be explored in future work.

G. Parallel Compile Time

As we define the switchboxes as reconfigurable, all the pages and switchboxes can be compiled in parallel on the cloud. The compile time for all the benchmarks are shown in Table VI. We can see the DW page compile time is a slightly longer than PS case, but still around 10–18 minutes. This is possibly because more partition pins are added between the pages and the connection boxes, which increase the routing difficulties. Nevertheless, we see the switchboxes' compile times are less than the page compile times. This means we can customize the interconnect without degrading the short compile time, as long as there are pages that also need to be recompiled. There are no switchbox (SWB) times for *digit recognition*, since all the page are feed-forward with systolic, nearest-neighbor connections.

TABLE VI: Application Mapping Time

Benchmark	SDSoC	Mono. (no BFT)	PS Leaves	Direct Wire Leaves	Direct Wire SWB
Rendering	1711	1495	606	737	603
Digit Reg	2569	2104	610	735	0
Spam Filter	1930	1780	568	695	593
Optical Flow	2997	2792	679	886	685
BNN	12001	11089	1004	1082	611
Face Detection	4136	2981	825	1089	606

H. Resource Overhead

In Tab. IV, we tabulate the resource overhead for our DW designs, in comparison with the PSNoC. We can see that our leaf interface overhead reduced by 47%–86% compared to the PSNoC. Initially, routing overhead is the total size of 8 switchbox pages. As user logic can also borrow the switchbox resource as hybrid pages, we reduce the initial overhead by the resource borrowed by user logic. We can see *face detection*'s routing overhead is small, since it is able to borrow more resource from the switchbox pages.

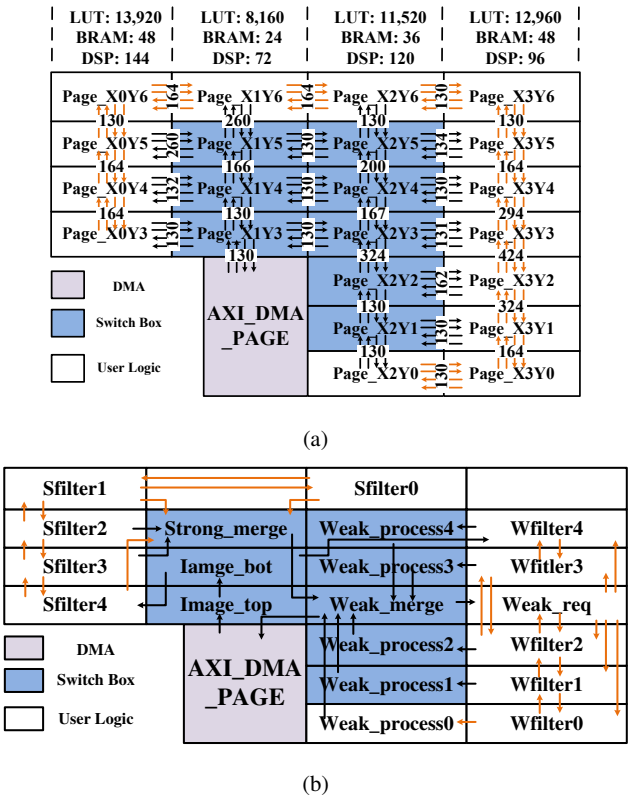

Fig. 11: (a) Overlay Floorplan. (b) Face Detection Mapping

VI. Discussion and Future Work

A. Fixed-Wiring Limitations

The pure DW gives up the complete virtualization of communication provided by the PSNoC. That is, the DW solution depends on designs not requiring more user ports than the overlay (Sec. V-D) can support. One solution is to serialize lower bandwidth ports (e.g., provide a logical 32b port with an 8b physical, DW path) to fit within pre-defined wiring constraints. Another is to consider a hybrid network with a PSNoC for fallback after exhausting high-speed DW capacity for the high throughput links.

When we encounter designs that require more ports than any overlay can support, this suggest the need for a new overlay. Given an iterative development style, we can fall back to the PSNoC, and this new overlay can be generated to support later revisions of the design. Ideally, custom overlay generation would be automated, so that a new, suitable overlay would become available in few hours.

B. Automatic Provisioning for Universal Overlay

As described, users can start with a default allocation of switches (e.g., Fig. 8, 11) and standard placement of compute pages and routing of switchbox pages will work. To embrace the universal pages, automation would be useful to select the allocation and placement of switchbox pages and automate the sharing of pages among computation and interconnect.

VII. Conclusion

Separation compilation and linking of FPGA designs can exploit parallelism to reduce mapping time without sacrificing the high inter-module bandwidth and low latency available on modern FPGAs. We show the inter-module linking problem can also be decomposed and performed in parallel with leaf page mapping. This linking can also be fast—comparable to the mapping time of logic—while exploiting the high, native FPGA wiring capacity. As a result, our fast mapped designs approach or exceed the performance of monolithic design mappings.

Acknowledgments

This work is funded in part by a Google Faculty Research Award and the Office of Naval Research under grant N000141812557. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Google or the Office of Naval Research. Xilinx donated Vivado tools for use in this work.

References

[1] Michael deLorimier, Nachiket Kapre, Nikil Mehta, Dominic Rizzo, Ian Eslick, Raphael Rubin, Tomás E. Uribe, Thomas F. Knight, Jr., and André DeHon. GraphStep: A system architecture for sparse-graph algorithms. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 143–151. IEEE, 2006.

[2] Dongjoon Park, Yuanlong Xiao, Nevo Magnezi, and André DeHon. Case for fast FPGA compilation using partial reconfiguration. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 235–2353. IEEE, 2018.

[3] Yuanlong Xiao, Dongjoon Park, Andrew Butt, Hans Giesen, Zhaoyang Han, Rui Ding, Nevo Magnezi, Raphael Rubin, and André DeHon. Reducing FPGA compile time with separate compilation for FPGA building blocks. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 153–161. IEEE, 2019.

[4] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, and Zhiru Zhang. Rosetta: A realistic high-level synthesis benchmark suite for software programmable FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 269–278, 2018.

[5] Xilinx. ZCU102 evaluation board. Accessed: 2019-06-12.

[6] Julien Boucaron, Anthony Coadou, and Robert De Simone. Latency-insensitive design: retry relay-station and fusion shell. *Electronic Notes in Theoretical Computer Science*, 245:23–33, 2009.

[7] Eylon Caspi. Design automation for streaming systems. Technical report, University of California at Berkeley, Dept of Electrical Engineering and Computer Science, 2005.

[8] Christopher Lavin, Marc Padilla, Jaren Lamprecht, Philip Lundrigan, Brent Nelson, and Brad Hutchings. HMFlow: Accelerating FPGA compilation with hard macros for rapid prototyping. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 117–124. IEEE, 2011.

[9] Christopher Lavin, Marc Padilla, Subhrashankha Ghosh, Brent Nelson, Brad Hutchings, and Michael Wirthlin. Using hard macros to reduce fpga compilation time. In *2010 International Conference on Field Programmable Logic and Applications*, pages 438–441. IEEE, 2010.

[10] Sen Ma, Zeyad Aklah, and David Andrews. Just in time assembly of accelerators. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 173–178, 2016.

[11] David Wilson and Greg Stitt. Seiba: An FPGA overlay-based approach to rapid application development. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE, 2019.

[12] Al-Shahna Jamal, Jeffrey Goeders, and Steven J Wilton. An FPGA overlay architecture supporting rapid implementation of functional changes during on-chip debug. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 403–4037. IEEE, 2018.

[13] Christopher Lavin, Marc Padilla, Jaren Lamprecht, Philip Lundrigan, Brent Nelson, and Brad Hutchings. RapidSmith: Do-it-yourself CAD tools for xilinx FPGAs. In *Proceedings of International Workshop on Field-Programmable Logic and Applications (FPL)*, September 2011.

[14] Chris Lavin and Alireza Kaviani. Rapidwright: Enabling custom crafted implementations for FPGAs. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 133–140. IEEE, 2018.

[15] Michael K Papamichael and James C Hoe. Connect: re-examining conventional wisdom for designing NoCs in the context of FPGAs. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 37–46, 2012.

[16] Stanford Concurrent VLSI Architecture Group. Open source Network-on-Chip router RTL. https://nocs.stanford.edu/cgi-bin/trac.cgi/wiki/Resources/Router. Accessed: 2010-09-30.

[17] Yutian Huan and André DeHon. FPGA optimized packet-switched NoC using split and merge primitives. In *2012 International Conference on Field-Programmable Technology*, pages 47–52. IEEE, 2012.

[18] Nachiket Kapre, Nikil Mehta, Michael Delorimier, Raphael Rubin, Henry Barnor, Michael J Wilson, Michael Wrighton, and Andre DeHon. Packet switched vs. time multiplexed FPGA overlay networks. In *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 205–216. IEEE, 2006.

[19] Nachiket Kapre and Jan Gray. Hoplite: A deflection-routed directional torus NoC for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 10(2):1–24, 2017.

[20] Nachiket Kapre. Deflection-routed butterfly fat trees on FPGAs. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2017.

[21] K. Vipin, J. Gray, and N. Kapre. Enabling partial reconfiguration and low latency routing using segmented FPGA NoCs. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, pages 1–8, 2017.

[22] Joseph Varghese, Michael Butts, and Jon Batcheller. An efficient logic emulation system. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1(2):171–174, June 1993.

[23] J. Babb, R. Tessier, M. Dahl, S. Z. Hanono, D. M. Hoki, and A. Agarwal. Logic emulation with virtual wires. *IEEE Transactions on Computed-Aided Design for Integrated Circuits and Systems*, 16(6):609–626, Jun 1997.

[24] Xilinx. Vivado design suite user guide partial reconfiguration. Accessed: 2018-12-05.

[25] Michael J Wirthlin and Brad L Hutchings. Sequencing run-time reconfigured hardware with software. In *Proceedings of the 1996 ACM fourth international symposium on Field-programmable gate arrays*, pages 122–128, 1996.

[26] Théodore Marescaux, Vincent Nollet, J-Y Mignolet, Andrei Bartic, Will Moffat, Prabhat Avasare, Paul Coene, Diederik Verkest, Serge Vernalde, and Rudy Lauwereins. Run-time support for heterogeneous multitasking on reconfigurable socs. *Integration*, 38(1):107–130, 2004.

[27] Mateusz Majer, Jürgen Teich, Ali Ahmadinia, and Christophe Bobda. The Erlangen Slot Machine: A dynamically reconfigurable FPGA-based computer. *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 47(1):15–31, 2007.

[28] Steve Young, Peter Alfke, Colm Fewer, Scott McMillan, Brandon Blodget, and Delon Levi. A high i/o reconfigurable crossbar switch. In *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003. FCCM 2003.*, pages 3–10. IEEE, 2003.

[29] Kahn Gilles. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.

[30] André DeHon, Yury Markovsky, Eylon Caspi, Michael Chu, Randy Huang, Stylianos Perissakis, Laura Pozzi, Joseph Yeh, and John Wawrzynek. Stream computations organized for reconfigurable execution. *Journal of Microprocessors and Microsystems*, 30(6):334–354, September 2006.

[31] Michael Butts, Anthony Mark Jones, and Paul Wasson. A structural object programming model, architecture, chip and tools for reconfigurable computing. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 55–64, April 2007.

[32] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computed-Aided Design for Integrated Circuits and Systems*, 20(9):1059–1076, 2001.

[33] Julien Boucaron, Jean-Vivien Millo, and Robert De Simone. Another glance at relay stations in latency-insensitive designs. *Electronic Notes in Theoretical Computer Science*, 146(2):41–59, 2006. Proceedings of the Second Workshop on Globally Asynchronous, Locally Synchronous Design (FMGALS 2005).

[34] Charles E Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE transactions on Computers*, 100(10):892–901, 1985.

[35] David Clark. Window and acknowledgement strategy in TCP. RFC 813, USC/ISI, Information Sciences Institute, University of Southern California, 4676 Admiralty Way, Marina del Rey, California, 90291, July 1982.

[36] M. Saldana, L. Shannon, J. S. Yue, S. Bian, J. Craig, and P. Chow. Routability of network topologies in FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(8):948–951, 2007.

[37] Xilinx. Zynq UltraScale+ MPSoC data sheet: Overview. Accessed: 2019-10-2.

[38] S. R. Dickey and R. Kenner. Combining switches for the nyu ultracomputer. In *Proceedings of The Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 521–523, 1992.

[39] Gaetano Borriello, Carl Ebeling, Scott Hauck, and Steven Burns. The triptych fpga architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 3(4):491–501, December 1995.

[40] Tom Kean and John Gray. Configurable hardware: Two case studies of micro-grain computation. *Journal of VLSI Signal Processing Systems for Signals, Image and Video Technology*, 2(1):9–16, 1990.

[41] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *XC6200 FPGA Advanced Product Specification*, version 1.0 edition, June 1996.

[42] Mingjie Lin. The amorphous FPGA architecture. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, pages 191—200, 2008.

[43] Yue Zha and Jing Li. Reconfigurable in-memory computing with resistive memory crossbar. In *Proceedings of the 35th International Conference on Computer-Aided Design*, pages 120:1–120:8, 2016.